Document Modelling

P. S. Langeslag

One-Hot Encoding

A one-hot vector encodes each token in a document as a sequence of zeroes and one `1`, in order to identify the token in a binary format. The number of digits in the series is equal to the token length of the document; the position of the `1` indicates the token's index (i.e. position).

The	1	0	0	0	0
tortoise	0	1	0	0	0
and	0	0	1	0	0
the	0	0	0	1	0
hare	0	0	0	0	1

- Advantages: no information is lost; the document can be reliably reconstructed.
- Disadvantages: memory requiments; each document requires its own model.

(See Lane et al. pp. 35–38.)

Bag of Words

A model storing information on each term and its frequency in a document, but discarding word order (and thus syntax).

```
>>> from collections import Counter
>>> tokens = ['the', 'tortoise', 'and', 'the', 'hare']
>>> Counter(tokens)
Counter({'the': 2, 'tortoise': 1, 'and': 1, 'hare': 1})
```

- Advantages: lower memory requirements; irrelevant words can be eliminated; words can be sorted by alphabet or frequency
- Disadvantage: loss of word order

(See Lane et al. pp. 38–41.)

Vector Space Model

- A vector is "an ordered list of numbers, or coordinates, in a vector space" (Lane 79).
- When used to represent word counts, each coordinate encodes the frequency of one term, and every new term thus adds a dimension in vector space (i.e. a new entry in the lexicon).
- For vectors to be comparable, we have to (1) normalize word frequency; and (2) use the same lexicon for all documents.
- To compare documents, we determine the cosine of the angle between their vectors; this is arrived at by calculating their **dot product** ($A \cdot B$); if the vectors differ in length, this figure should be normalized by dividing by the product of their lengths: $\cos\theta = \frac{A \cdot B}{|A||B|}$

Dot Product

The sum of products between corresponding entries in two sequences of numbers: $\sum_{i=1}^{n} a_i b_i$

Thus given a query containing four terms `"deofol"`, `"helle"`, `"ancor"`, `"þunorrad"`

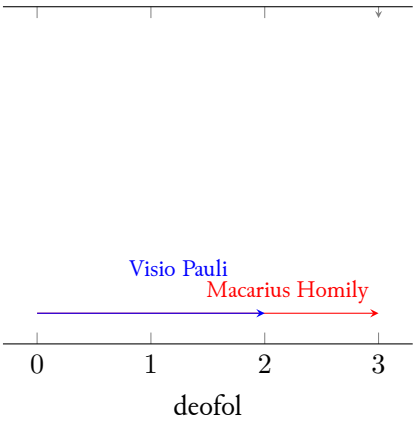
```
>>> import numpy
>>> query = numpy.array([1, 1, 1, 1])
>>> textm = numpy.array([1, 1, 1, 1])
>>> textp = numpy.array([1, 1, 0, 0])
```

the dot product between `query` and `textm` would be $1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 = 4$, whereas `textp` would come to only `2`:

```
>>> query.dot(p)
2
```

Thus the dot product allows us to calculate the overlap between bags of words.

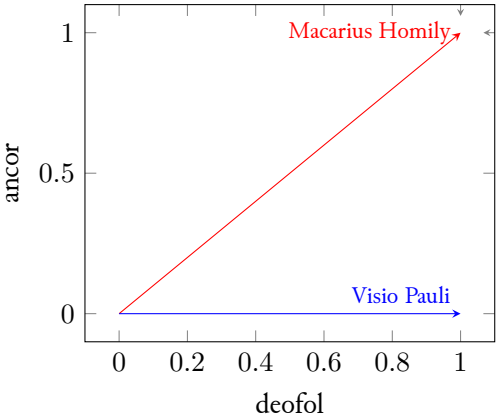
Example: A One-Dimensional Vector Space



(Using absolute counts)

```
>>> query = numpy.array([1])
>>> textm = numpy.array([3])
>>> textp = numpy.array([2])
```

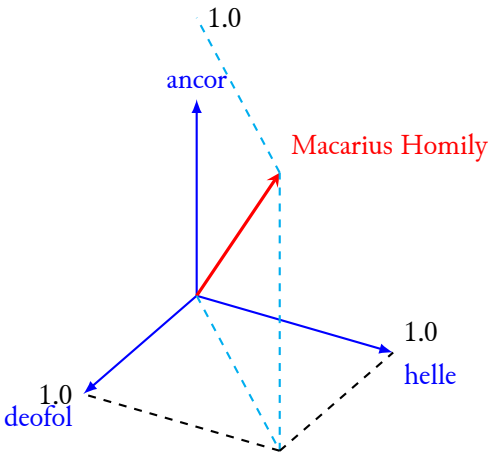
Example: A Two-Dimensional Vector Space



(Using binary switches)

```
>>> query = numpy.array([1, 1])
>>> textm = numpy.array([1, 1])
>>> textp = numpy.array([1, 0])
```

Example: A Three-Dimensional Vector Space



(Using binary switches)

```
>>> query = numpy.array([1, 1, 1])
>>> textm = numpy.array([1, 1, 1])
```

Example: A Four-Dimensional Vector Space

(Using binary switches)

Given four dimensions `"deofol"`, `"helle"`, `"ancor"`, `"þunorrad"`

```
query = [1, 1, 1, 1]
textm = [1, 1, 1, 1]
textp = [1, 1, 0, 0]
```

Zipf's Law

A word's frequency in a natural corpus *f(r)* is inversely proportional to its rank (*r*) in the word frequency table.

Zipf's Law

A word's frequency in a natural corpus *f(r)* is inversely proportional to its rank (*r*) in the word frequency table.

$$f(r) \propto \frac{1}{(r+\beta)^{\alpha}}$$

where $\alpha \approx 1$ and $\beta \approx 2.7$

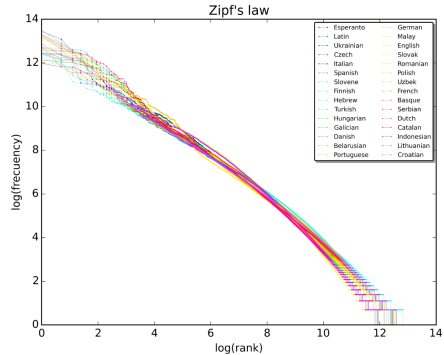


Figure 1: Frequency/rank log plot for the first 10 mln words in 30 Wikipedias (CC-BY-SA Sergio Jimenez)

Logarithm

- The logarithm of a number x is the inverse to its exponent.
- It equals the exponent by which the base must be raised to yield x, i.e. $b^y = x$
- Adding up the logarithms of two or more numbers yields the logarithm of their product. *This makes logarithmic arithmetic computationally frugal.*
- Logarithmic functions are used in NLP to represent word frequencies on a linear scale (see previous slide).
- Like exponentiation, the logarithm requires that a base b be specified, but for our purposes the chosen base doesn't matter as long as it's constant.
- The notation of the logarithm is $log_b(x)$ for antilogarithm x to base b. (Base 10 is assumed if you leave out $_b$.)

TF-IDF

- **Term frequency** is a term's frequency in a document, whether in absolute numbers or divided by its total token count ("normalized term frequency").
- **Inverse document frequency** is the total document count divided by the number of documents containing your term; but to normalize Zipf distribution, we take the logarithm of this figure.
- **TF-IDF** is the product of these two numbers, indicating a term's statistical importance in a single document as judged by its prevalence in the corpus as a whole.
- (If we carry out *all* calculations in the service of TF-IDF in log space, we can add and subtract instead of multiply and divide; this is a convenient feature of logarithms.)
- We can speak of *statistical importance*, not just *frequency*, because TF-IDF measures how exceptional a term's frequency in a document is as compared against the corpus as a whole.

Bibliography

Bird, Steven, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. Sebastopol, CA: O'Reilly, 2009. https://www.nltk.org/book/.

Eisenstein, Jacob. *Introduction to Natural Language Processing*. Cambridge, MA: MIT Press, 2019.

Jurafsky, Dan, and James H. Martin. *Speech and Language Processing*. 3rd ed. draft., 2021. http://web.stanford.edu/~jurafsky/slp3/.

Lane, Hobson, Cole Howard, and Hannes Hapke. *Natural Language Processing in Action: Understanding, Analyzing, and Generating Text with Python*. Shelter Island, NY: Manning, 2019.

Matthes, Eric. *Python Crash Course*. 2nd ed. San Francisco, CA: No Starch, 2019.

Python Software Foundation. "Python," October 4, 2020. https://www.python.org/.

Vasiliev, Yuli. *Natural Language Processing Using Python and spaCy: A Practical Introduction*. San Francisco: No Starch Press, 2020.